Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

An Analysis of State of the Art Tools for Preparing DL_POLY_4 for Exascale

Buket Benek Gursoy [a*], Henrik Nagel [b]

[a]*Irish Center for High End Computing, Ireland,* [b]*Norwegian University of Science and Technology, Norway*

Abstract

This whitepaper investigates the potential benefit of using the OpenACC directive-based programming tool for enabling DL_POLY_4 on GPUs. DL_POLY is a well-known general-purpose molecular dynamics simulation package, which has already been parallelised using MPI-2. DL_POLY_3 was accelerated using the CUDA framework by the Irish Centre for High-End Computing (ICHEC) in collaboration with Daresbury Laboratory. In this work, we have been inspired by the existing CUDA port to evaluate the effectiveness of OpenACC in further enabling DL_POLY_4 on the road to Exascale. We have been particularly concerned with investigating the benefits of OpenACC in terms of maintainability, programmability and portability issues that are becoming increasingly challenging as we advance to the Exascale era. The impact of the OpenACC port has been assessed in the context of a change in the reciprocal vector dimension for the calculation of SPME forces. Moreover, the interoperability of OpenACC with the existing CUDA port has been analysed.

1. Introduction

The DL_POLY package [1] is a collection of parallel programs and data files, designed for large-scale molecular dynamics simulations at the STFC Daresbury Laboratory by I.T. Todorov and W. Smith. It was originally developed for the molecular simulation community CCP5 in the United Kingdom in the mid-1990s [1, 2]. Three versions of DL_POLY are currently available: DL_POLY_2, which has been developed using Replicated Data parallelisation strategy [3], DL_POLY_3, which has been parallelised based on Domain Decomposition model [4] and DL_POLY_4, which is an extension of the previous version with a new parallel I/O and netCDF support [1]. A full specification of these can be found in the DL_POLY User Manual [5]. DL_POLY_4 is available worldwide under an STFC licence that is free of cost to academic scientists for non-commercial research. DL_POLY has been widely-used in broad range applications of molecular dynamics since it was first officially released [2]. In particular, it has effectively exploited PRACE Tier-1 and Tier-0 systems, and has been used across different projects in the PRACE RI [6, 7, 8, 9].

DL_POLY_4 was written in modularised Fortran90 with MPI-2. It scales up to many thousands of CPU cores [2]. In collaboration with Daresbury Labs, ICHEC was involved in porting DL_POLY version 3.10 to GPUs within the PRACE-1IP and -2IP FP7 projects. GPGPU acceleration of DL_POLY_3 was implemented using CUDA and combined with OpenMP [6]. A significant speed-up was obtained per accelerated component for the test case TEST2 on Fermi architecture. Subsequently, the original GPU port was updated to align with the changes in DL_POLY version 4.01.1 in [7]. When running DL_POLY across 16 GPU nodes (with 2 NVIDIA Tesla M2090s per node), there was an approximate 30% reduction in the wall-clock time. Other works carried out within PRACE in the context of GPU acceleration of DL_POLY can be found in [8, 9]. Recently, DL_POLY version 4.05.1 has been released with a number of modifications in the source code. The CUDA port of DL_POLY is no longer up to date with those changes and fails to produce correct results for certain test cases.

* Corresponding author. *E-mail address*: buket.gursoy@ichec.ie

Although CUDA provides a significant performance improvement, it is hard to maintain the code in order to keep up with the changes of the vanilla MPI code. This has inspired us to consider an alternative among more easily maintainable and less intrusive programming models that have relevance on the road to Exascale compared to the CUDA framework. As such in this paper, we evaluate the benefits of implementing OpenACC in DL_POLY to mirror further changes in the source code and to assess the potential of it for future Exascale architectures.

1.1. OpenACC Programming Model

The OpenACC API is a high-level directive-based programming model for heterogeneous systems. Similar to the execution model of CUDA, it is designed for accelerating compute-intensive loops and regions of C/C++/Fortran code by offloading to the GPUs. Parallelism is exploited through a set of compiler directives only by adding a number of lines to the source code. It was first released in November 2011 and the latest version OpenACC 2.0 was released in August 2013 [10]. It was developed by CAPS, CRAY, PGI and NVIDIA and supported by compilers from PGI, CRAY and CAPS [10].

There are several large-scale applications successfully ported to GPUs with OpenACC in areas of linear algebra, engineering, medicine, computational fluid dynamics and hydrodynamics [11-15]. In particular, application of OpenACC within the S3D code, which is one of the early experiences with OpenACC at the ORNL, has proven OpenACC as a promising tool for future Exascale systems [12]. Moreover, as highlighted in the PRACE-3IP Deliverable D7.2.1 "A report on the Survey of HPC Tools and Techniques", OpenACC has been receiving an increasing attention for next generation HPC systems [16]. Motivated by these, the work presented here is an early evaluation of the effectiveness of OpenACC for enabling DL_POLY on many core architectures and investigating the challenges for Exascale computing.

1.2. Structure of Whitepaper

The layout of the paper is as follows: Section 2 gives a brief description of the system used to obtain the results presented in this paper. It also shows profiling of DL_POLY that investigates the most time consuming parts. Section 3 describes the `ewald_real_forces` subroutine and Section 4 describes the `ewald_spme_forces` subroutine. Furthermore, performance results and the impact of using different `gang` and `vector` parameters are presented in these two sections. Section 5 is an analysis of our approach of integrating OpenACC with CUDA for the `link_cell_pairs_remove_exclusions` component. Finally, Section 6 is a brief conclusion of the paper.

2. Setup and Profiling

2.1. Test Setup

All tests were conducted on the Abel supercomputer located at the University of Oslo, Norway. The Abel computing cluster consists of more than 650 Supermicro X9DRT compute nodes each having two Intel E5-2670 Sandy Bridge 2.6 GHz CPUs and 64 GBs of Samsung DDR3 memory. A set of nodes is equipped with NVIDIA K20 GPUs and Intel Xeon Phi. There is an FDR Infiniband connection between all nodes. It was ranked as 96 in the Top500 list in June 2011. For more details on the hardware specifications, see [17].

OpenACC was enabled using PGI v13.9 and OpenACC tests were run on the Intel E5-2670 and an NVIDIA Tesla K20 GPU. The implementation in Section 3 was compiled using Open MPI 1.7.2 and PGI v13.9 while the implementation in Section 4 was compiled using MPICH-3.0.4 and PGI v13.9. The CUDA kernels were compiled with NVIDIA CUDA 5.5.

The data set used was the TEST2 test case from the data sets that come with DL_POLY. It is a configuration file generated for Sodium Chloride with 216000 ions. The first thing to notice is that in the CONTROL file there is a field called "job time". This is initially set to 600 seconds, meaning that the simulation will stop normally, but with a wrong result after at most 600 seconds. This can occur when using OpenACC, because one then must reduce the number of processes to the number of accelerator cards used, in this case one. The field "job time" must therefore be increased, if one wants the simulation to end after a certain number of iterations, instead of after a certain time.

2.2. Profiling Results

The serial version of DL_POLY_4.05.1 was first profiled using the default PGI profiler pgprof in Figure 1. It shows that `ewald_real_forces` is the second most time-consuming and `spme_forces` is the fourth most time-consuming component of the execution. The `spme_forces` subroutine is called within the `ewald_spme_forces` subroutine. This work focuses on these two subroutines for an early evaluation of the effect of OpenACC for DL_POLY. Note that they were also considered in the original CUDA port of DL_POLY [6]. Briefly, `ewald_sum` is one of the approaches in DL_POLY used to compute the long-ranged electrostatic potentials [2, 5]. The calculation has three main steps:

i. Real space evaluation;
ii. Reciprocal space evaluation;
iii. Self-energy correction.

The real space contributions are calculated by the `ewald_real_forces` routine while the reciprocal space contributions are calculated by the `ewald_spme_forces` routine [5].

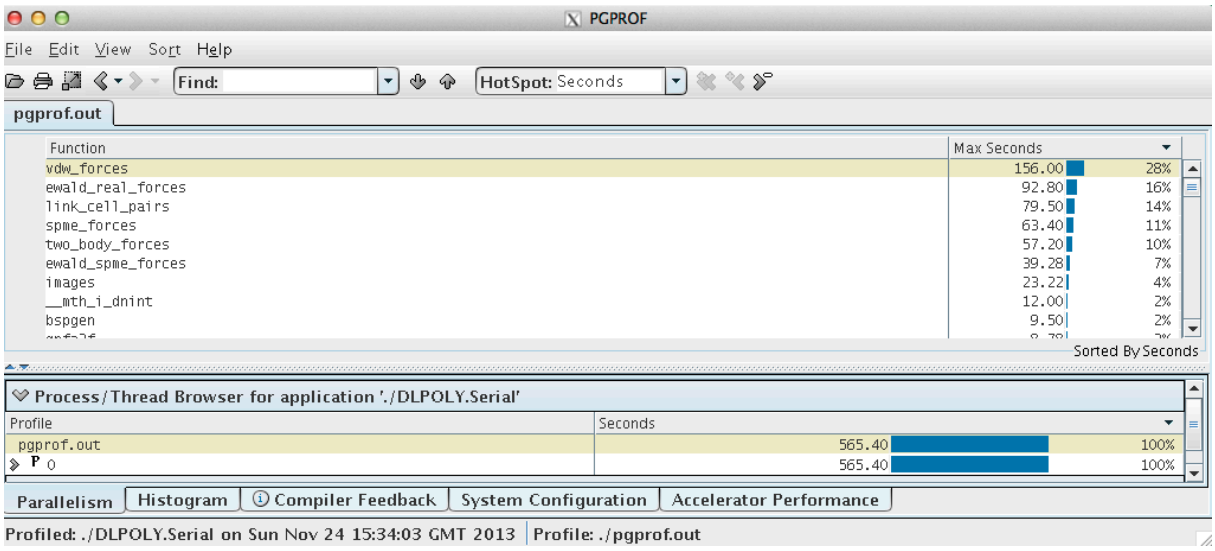

Figure 1 – Profiling of serial DL_POLY_4.05.1 for TEST2 using pgprof

To further detect hotspots in these routines, Allinea MAP v4.2 profiler was used. For this purpose, DL_POLY was compiled with Intel compilers 2013-sp1 on the ICHEC HPC cluster Fionn [18]. Figures 2-4 illustrate sections from the profiling results. Specifically, Figure 2 lists the first four most time-consuming functions of DL_POLY. These were called within the `two_body_forces` routine. Figure 3 shows that 15.6% of the overall execution time was spent in the main do-loop of `ewald_real_forces`, which was 15.9% of the overall time. Similarly, Figure 4 shows that 6.5% of the overall execution time was spent in the main do-loop of `ewald_spme_forces`, which was 15.1% of the overall time.

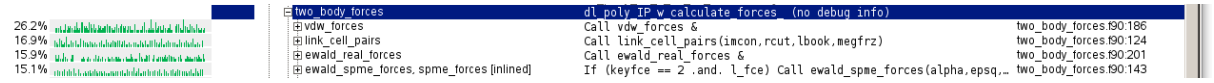

Figure 2 – Profiling of serial DL_POLY_4.05.1 for TEST2 using Allinea MAP

Figure 3 – Hotspot in `ewald_real_forces`

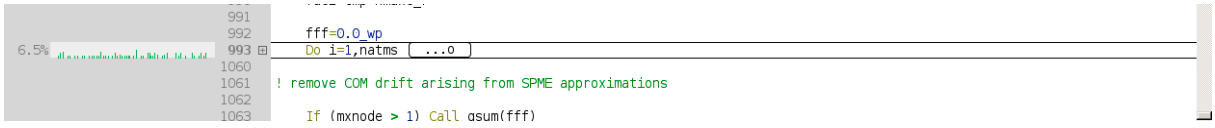

Figure 4 – Hotspot in `ewald_spme_forces`

3. Ewald REAL Forces

3.1. Initial observations

The subroutine `ewald_real_forces` is called 21816000 times and about 93 seconds out of the total 551 seconds is spent in this subroutine, without using OpenACC. The loop in `ewald_real_forces` has with the test data TEST2, between 42 and 209 iterations. Because of the very large number of times that this subroutine is called combined with the low number of iterations, this function is not a good candidate for OpenACC optimisation.

If we simply put an "`!$acc kernels loop`" pragma before the loop in `ewald_real_forces` and then compile with OpenACC enabled, then we get the following warning from the pgif90 compiler:

```
104, Complex loop carried dependence of 'fxx' prevents parallelization
     Loop carried dependence due to exposed use of 'fxx(:)' prevents parallelization
     Complex loop carried dependence of 'fyy' prevents parallelization
     Loop carried dependence due to exposed use of 'fyy(:)' prevents parallelization
     Complex loop carried dependence of 'fzz' prevents parallelization
     Loop carried dependence due to exposed use of 'fzz(:)' prevents parallelization
     Accelerator scalar kernel generated
```

This increases the total execution time from approximately 551 seconds to 140 hours (estimated). However, the results that are produced by the OpenACC parallelised code are also wrong. This can be seen in the STATIS file that DL_POLY updates regularly during the execution. The numbers in this file are clearly very different from the numbers that DL_POLY produces without the OpenACC pragma.

3.2. First approach

There are initially three problems to solve:

i. DL_POLY produces wrong results with OpenACC;
ii. With the TEST2 dataset, the loop in `ewald_real_forces` is called 21816000 times, which means that data must be moved to and from the accelerator card that many times;
iii. When we use the "`!$acc kernels loop`" pragma, the arrays fxx, fyy and fzz prevent the loop in `ewald_real_forces` from being parallelised.

The first problem is very serious. DL_POLY simply produces wrong results with OpenACC without giving any indication of what might be the cause of this. During the project, large amounts of time were spent on trying to find the cause of this problem, since there is obviously no point in optimizing a program that produces wrong results. Since a scalar kernel was generated, it is strange that this scalar kernel produces wrong results. The data was also examined in a debugger to see if multiple OpenACC threads (if any) updated the same array elements, but this did not seem to be the case. In the end, solving the third problem also solved this one. We believe that the reason is related to the data dependencies among used arrays between successive iterations.

The second problem can be alleviated by noticing that `ewald_real_forces` is called from inside of an outer loop in two_body_forces.f90. It is called only when a variable keyfce is 2 and the value of keyfce does not change the value during the execution of the loop. It is therefore possible to check if keyfce is 2 before the execution of the outer loop in `two_body_forces` is entered and thereby create a special outer loop for `ewald_real_forces`. The outer loop can thus be moved into `ewald_real_forces`, so that OpenACC code for moving data to the accelerator card can be placed before the execution of the outer loop is entered.

The third problem can be solved by noticing that fxx, fyy, and fzz are updated in the following manner:

```
fx = egamma*xdf(m)
fy = egamma*ydf(m)
fz = egamma*zdf(m)
fxx(jatm)=fxx(jatm)-fx
fyy(jatm)=fyy(jatm)-fy
fzz(jatm)=fzz(jatm)-fz
```

The arrays xdf, ydf, and zdf do not change during the execution of the loop in `ewald_real_forces`, so if we keep track of the produced egamma values, then we can update fxx, fyy, and fzz in a loop running on the CPU after the loop running on the accelerator card has finished.

These changes reduce the total run-time from an estimated 140 hours to 13 hours, which is a significant improvement but still far from the 540 seconds that the original code took when running on one process without OpenACC. A major improvement of the code is that it now returns the same results as the original code produces. Timing results are presented in Table 1 when running on one process and one accelerator card. The following methods are used to measure the time spent for the OpenACC port of `ewald_real_forces` function:

i. `MPI_Wtime()` function at the start and the end of the `ewald_real_forces` function;
ii. PGI compiler's timing enabled by setting the environment flag `PGI_ACC_TIME=1`.

Timing method	Run time
`MPI_Wtime()`, OpenACC disabled	318.7 sec
`MPI_Wtime()`, OpenACC enabled	46575.2 sec
`PGI_ACC_TIME`, OpenACC enabled	13298.9 sec

Table 1 - Runtimes of the first approach in `ewald_real_forces` for different timing methods

When OpenACC has been disabled, the time spent in the `ewald_real_forces` function is only of 318.7 seconds. When OpenACC is enabled the total run-time for DL_POLY becomes 13 hours and `MPI_Wtime()` reports that of these 12.9 hours were spent in this function. The PGI compiler's timing reports state that the OpenACC part of the code only took 3.7 hours to run. By investigating the difference between `MPI_Wtime()` and `PGI_ACC_TIME`, it looks like PGI profiler doesn't include all overhead coming from the OpenACC parallelisation and GPU initialisation. Thus, we prefer to use `PGI_ACC_TIME` to compare the execution time of the same region between different OpenACC implementations only.

3.3. Second approach

The next logical step is to reduce the number of if-statements in the kernel code. The first step in this process is to create an index list on the CPU that only contains the indices for which the OpenACC loop makes it past the first two initial if-statements. Using this index list on the accelerator card, the first two if-statements can be removed. The loop does not produce any results for indices that do not make it past these two if-statements. This leaves the code in the loop with only a single if-statement. The effect of this small improvement is illustrated in Table 2.

Timing method	Run time
`MPI_Wtime()`, OpenACC disabled	320.9 sec
`MPI_Wtime()`, OpenACC enabled	39817.0 sec
`PGI_ACC_TIME`, OpenACC enabled	13032.1 sec

Table 2 - Runtimes of the second approach in `ewald_real_forces` for different timing methods

3.4. Third approach

In order to remove the last if-statement from the loop, it is necessary to store more temporary results in the loop. That way, the GPU code can always do the interior of the last if-statement, while the CPU then only updates the result variables, in the case where the if-statement is executed. Doing this also removes the reduction variables from the loop and moves the update of these variables to the CPU. In particular, the moving of the update of the reduction variables to the CPU results in a significant performance increase over the last iteration of the code. The pseudo codes of the main loop for both the serial version and OpenACC port are illustrated in Table 3.

```
Do m=1,list(0,iatm)
  …
  If (Abs(chgprd) > zero_plus) Then
  chgprd=chgprd*chgea
  rsq=rsqdf(m)
    If (rsq < rcsq) Then
    rrr = Sqrt(rsq)
    k   = Int(rrr*rdrewd)
    ppp = rrr*rdrewd - Real(k,wp)

    gk0 = fer(k)
    gk1 = fer(k+1)
```

```
Integer, Dimension(:), Allocatable, save :: m_list
Real(Kind=wp), Dimension(:,:), Allocatable, save ::
results

!$acc data copyin(list, chge, rdrewd, fer, erc)
Do iatm=1,natms
  limit=list(0,iatm)
  Do m=1,limit
    …
    If (Abs(chgprd) > zero_plus) Then
      rsq=rsqdf(m)
      If (rsq < rcsq) Then
```

```
    gk2 = fer(k+2)

    t1 = gk0 + (gk1-gk0)*ppp
    t2 = gk1 + (gk2-gk1)*(ppp-1.0_wp)

    egamma      =      (t1      +      (t2-
t1)*ppp*0.5_wp)*chgprd
    fx = egamma*xdf(m)
    fy = egamma*ydf(m)
    fz = egamma*zdf(m)

    fix=fix+fx
    fiy=fiy+fy
    fiz=fiz+fz

    If (jatm <= natms) Then
      fxx(jatm)=fxx(jatm)-fx
      fyy(jatm)=fyy(jatm)-fy
      fzz(jatm)=fzz(jatm)-fz
    End If

    If   (jatm   <=   natms   .or.   idi   <
ltg(jatm)) Then
      …

      engcpe_rl = engcpe_rl + (t1 + (t2-
t1)*ppp*0.5_wp)*chgprd
      vircpe_rl = vircpe_rl - egamma*rsq

      strs1 = strs1 + xdf(m)*fx
      strs2 = strs2 + xdf(m)*fy
      strs3 = strs3 + xdf(m)*fz
      strs5 = strs5 + ydf(m)*fy
      strs6 = strs6 + ydf(m)*fz
      strs9 = strs9 + zdf(m)*fz
    End If
    End If
  End If
End Do
```

```
        m_list_size = m_list_size + 1
        m_list(m_list_size) = m
      End If
    End If
  End Do

!$acc kernels loop present(list, chge, rdrewd, fer,
erc)
  Do j=1,list(0,iatm)
    m = m_list(j)
    jatm=list(m,iatm)
    chgprd=chge(jatm)

    chgprd=chgprd*chgea
    rsq=rsqdf(m)
    rrr = Sqrt(rsq)
    k   = Int(rrr*rdrewd)
    ppp = rrr*rdrewd - Real(k,wp)
    gk0 = fer(k)
    gk1 = fer(k+1)
    gk2 = fer(k+2)
    t1 = gk0 + (gk1-gk0)*ppp
    t2 = gk1 + (gk2-gk1)*(ppp-1.0_wp)

    egamma = (t1 + (t2-t1)*ppp*0.5_wp)*chgprd
    results(j,1) = egamma
    …
    results(j,2) = (t1 + (t2-t1)*ppp*0.5_wp)*chgprd
    results(j,3) = egamma*rsq
  End Do
!$acc end loop

  Do j=1,m_list_size
    m = m_list(j)
    jatm   = list(m,iatm)
    chgprd = chge(jatm)

    fx = results(j,1)*xdf(m)
    fy = results(j,1)*ydf(m)
    fz = results(j,1)*zdf(m)

    fix=fix+fx
    fiy=fiy+fy
    fiz=fiz+fz

    If  (Abs(chgprd)>zero_plus  .and.  rsqdf(m)<rcsq)
Then
      If (jatm<=natms) Then
        fxx(jatm)=fxx(jatm)-fx
        fyy(jatm)=fyy(jatm)-fy
        fzz(jatm)=fzz(jatm)-fz

        engacc2 = engacc2 + results(j,2)
        viracc2 = viracc2 - results(j,3)

        strs1 = strs1 + xdf(m)*fx
        …
      Else If (idi < ltg(jatm)) Then
        engacc2 = engacc2 + results(j,2)
        viracc2 = viracc2 - results(j,3)

        strs1 = strs1 + xdf(m)*fx
        …
      End If
    End If
  End Do
End Do
!$acc end data
```

Table 3 – Sketch of the main loop in `ewald_real_forces` and its OpenACC port

Performance results are listed in Table 4. Execution times with respect to different number of gangs and vector length are also included in the table. In particular, timing data provided by PGI for the (`gang`(128) `vector`(128)) are as follows:

```
Accelerator Kernel Timing data
/cluster/home/hrn/prace/dl_poly_4.05_acc/source/ewald_real_forces.f90
  ewald_real_forces  NVIDIA  devicenum=0
    time(us): 976,170,691
    123: data region reached 101 times
      123: data copyin reached 3636 times
          device time(us): total=9,083,435 max=2,843 min=7 avg=2,498
    215: compute region reached 21816000 times
      215: data copyin reached 43632000 times
          device time(us): total=363,767,240 max=192 min=5 avg=8
      216: kernel launched 21816000 times
          grid: [128]  block: [128]
          device time(us): total=381,256,645 max=1,575 min=12 avg=17
          elapsed time(us): total=587,035,174 max=1,587 min=24 avg=26
      295: data copyout reached 21816000 times
          device time(us): total=222,063,371 max=1,756 min=7 avg=10
```

Timing method	**Run time**
`MPI_Wtime()`, Serial, original code	308.6 sec
`MPI_Wtime()`, OpenACC disabled	336.9 sec
`PGI_ACC_TIME`, OpenACC enabled (default)	979.5 sec
`MPI_Wtime()`, OpenACC (default)	2695.0 sec
`MPI_Wtime()`, OpenACC(`gang`(128) `vector`(32))	2672,6 sec
`MPI_Wtime()`, OpenACC (`gang`(32) `vector`(128))	2643.0 sec
`MPI_Wtime()`, OpenACC (`gang`(64) `vector`(128))	2616.1 sec
`MPI_Wtime()`, OpenACC(`gang`(256) `vector`(128))	2564.9 sec
`MPI_Wtime()`, OpenACC(`gang`(128) `vector`(128))	2539.7 sec

Table 4 - Runtimes of the third approach in `ewald_real_forces` for different timing methods and OpenACC implementations

4. Ewald SPME Forces

The main loop of the `spme_forces` subroutine is illustrated in Table 5. There is only one data dependency for the calculation of the columbic forces of the interacting atoms that can be handled when the force terms are reduced at the end of the loop.

4.1. OpenACC Port

Data dependency carried across the i^{th} iteration space was handled by introducing the reduction parameters corresponding to the elements in the forces array of dimension four. The OpenACC parallelisation was performed using a kernels loop directive. The following approaches were applied:

i. Outermost loop was parallelised using a `gang` clause across the y^{th} grid dimension (blockIdx.y);

ii. The l-loop was parallelised using a `vector` clause across the y^{th} block dimension (threadIdx.y);

iii. The j-loop was parallelised using a `vector` clause across the x^{th} block dimension (threadIdx.x).

Note that the number of gangs and vector lengths were chosen inspired by the CUDA port of `spme_forces`. As it can be seen from the information messages below, the PGI compiler handled necessary data copies. A small change in the source was to remove the main computation from the "if branching". As a result, four OpenACC directives were added while the corresponding CUDA port consists of approximately 500 lines of CUDA code.

```
spme_forces:
  1391, Generating present_or_copyin(chge(1:natms))
        Generating present_or_copy(fcx(1:natms))
```

```
    Generating present_or_copy(fcy(1:natms))
    Generating present_or_copy(fcz(1:natms))
    Generating present_or_copyin(bsdz(:mxspl,:natms))
    Generating present_or_copyin(bspz(:mxspl,:natms))
    Generating present_or_copyin(izz(:natms))
    Generating present_or_copyin(bsdy(:mxspl,:natms))
    Generating present_or_copyin(bspy(:mxspl,:natms))
    Generating present_or_copyin(iyy(:natms))
    Generating present_or_copyin(ixx(:natms))
    Generating present_or_copyin(qqc_domain(:,:,:))
    Generating present_or_copyin(bsdx(:mxspl,:natms))
    Generating present_or_copyin(bspx(:mxspl,:natms))
    Generating present_or_copyin(rcell(:))
    Generating present_or_copy(fxx(1:natms))
    Generating present_or_copy(fyy(1:natms))
    Generating present_or_copy(fzz(1:natms))
    Generating NVIDIA code
    Generating compute capability 1.3 binary
    Generating compute capability 2.0 binary
    Generating compute capability 3.0 binary
1392, Loop is parallelizable
    Accelerator kernel generated
  1392, !$acc loop gang(900) ! blockidx%x
  1403, !$acc loop vector(8) ! threadidx%y
  1416, !$acc loop vector(8) ! threadidx%x
1403, Loop is parallelizable
1409, Loop is parallelizable
1416, Loop is parallelizable
```

```
fff=0.0_wp

Do i=1,natms
  tmp=chge(i)

  If (Abs(tmp) > zero_plus) Then
  ! initialise forces
  …
    Do l=1,mxspl
    …
      Do k=1,mxspl
      …
        Do j=1,mxspl
        …
        End Do
      End Do
    End Do
    ! accumulate forces
    fff(0)=fff(0)+1.0_wp
    fff(1)=fff(1)+fx
    fff(2)=fff(2)+fy
    fff(3)=fff(3)+fz
    ! load forces
    …
    ! infrequent calculations copying
    …
   End If
End Do
```

```
Real ( Kind = wp ) :: fff0, fff1, fff2, fff3

fff0=0.0_wp
…
!$acc    kernels    loop    gang(900)    private(tmp)
reduction(+:fff0,fff1,fff2,fff3)
Do i=1,natms
  tmp=chge(i)

  If (Abs(tmp) <= zero_plus) cycle

  ! initialise forces
  …
  !$acc loop vector(8)
  Do l=1,mxspl
  …
    Do k=1,mxspl
    …
      !$acc loop vector(8)
      Do j=1,mxspl
      …
      End Do
    End Do
  End Do
  ! accumulate forces
  fff0=fff0+1.0_wp
  fff1=fff1+fx
  fff2=fff2+fy
  fff3=fff3+fz
  ! load forces
  …
  ! infrequent calculations copying
  …
End Do
!$acc end kernels

fff(0)=fff(0)+fff0
fff(1)=fff(1)+fff1
fff(2)=fff(2)+fff2
fff(3)=fff(3)+fff3
```

Table 5 – Sketch of the main loop in `spme_forces` and its OpenACC port

4.2. Performance Results

For the results of this section, nfold parameter of TEST2 was set to (4,4,4) in the CONTROL input file in order to increase the size of the calculation. The following methods are used to measure the time spent for the main loop of `spme_forces` function:

i. `system_clock()` function at the start and the end of the main loop;
ii. PGI compiler's timing enabled by setting the environment flag `PGI_ACC_TIME=1`.

First, the serial runtime was measured as 60.2 sec. Then, it was compiled after simply adding "`!$acc kernels loop`" with the reduction operation described in the previous section. The automatic PGI compiler values provided a vector length of 128, which means that the iterations of the loop are broken into vectors of 128. The execution time was 1.5x faster than the serial version. To assess the impact of different loop scheduling, a number of gangs were taken as 64 and vector length was taken as 256. However, it performed worse than the serial execution (See Table 6.). When the CUDA-inspired values were taken, approximately 8.8x faster execution time was obtained. It performs ~83% better than the compiler generated version. Since the updated CUDA code of `spme_forces` in [7] works correctly for DL_POLY_4.05.1, the time for the execution of CUDA version was also measured (See Table 7). As a result, the best OpenACC performance can be achieved when the CUDA-inspired parameters are used. Still, it doesn't perform better than the CUDA port.

Implementation	**Runtime**
`PGI_ACC_TIME`, OpenACC (default)	39.321 sec
`PGI_ACC_TIME`, OpenACC (gang(64) vector(256))	71.653 sec
`PGI_ACC_TIME`, OpenACC(gang(900) vector(8x8))	6.866 sec

Table 6 – Runtimes of the main loop in `spme_forces` for different implementations of OpenACC

Implementation	**Runtime**
`system_clock()`, Serial, original code	60.2 sec
`system_clock()`, OpenACC(gang(900) vector(8x8))	12.45 sec
`system_clock()`, CUDA	2.4271 sec

Table 7 – Runtimes of the main loop in `spme_forces` for different implementations

Below is the kernel timing data for the best OpenACC implementation. It shows that approximately 1.8 sec. was spent for copying data from host to device, 4.7 sec. were spent for kernel execution, 0.03 sec. was spent for the reduction operation and 0.08 sec. was spent for copying data back to the CPU memory.

```
Accelerator Kernel Timing data
/cluster/home/bgursoy/DLPOLY/dl-poly/source/ewald_spme_forces_cuda.f90
  spme_forces  NVIDIA  devicenum=0
    time(us): 6,865,577
    1378: compute region reached 101 times
      1378: data copyin reached 1515 times
          device time(us): total=1,765,135 max=2,318 min=6 avg=1,165
      1379: kernel launched 101 times
        grid: [900]  block: [8x8]
          device time(us): total=4,748,412 max=201,168 min=44,244 avg=47,013
          elapsed time(us): total=4,750,146 max=201,188 min=44,265 avg=47,031
      1379: reduction kernel launched 101 times
        grid: [4]  block: [256]
          device time(us): total=270,836 max=13,647 min=2,023 avg=2,681
          elapsed time(us): total=272,546 max=13,663 min=2,039 avg=2,698
      1445: data copyout reached 303 times
          device time(us): total=81,194 max=317 min=265 avg=267
```

In the CUDA port, 0.84 sec. was spent for copying data from host to device, 1.55 sec. was spent during kernel execution and 0.039 sec. was spent for copying data back to the CPU memory. The kernel execution is

approximately 2.8x slower in OpenACC. However, it is more efficient in terms of easy programming and code maintainability.

Figure 5 illustrates the performance of the main loop of the OpenACC version and serial version using different lattice vector dimensions. The OpenACC version scales well when the dimension increases. However, data transfer overhead between the CPU and GPU memory dominates the overall execution time heavily for bigger dimensions. For instance, when the dimension was 106, 26% of the runtime spent during data transfer. When it was 212, data transfer was 63% of the runtime.

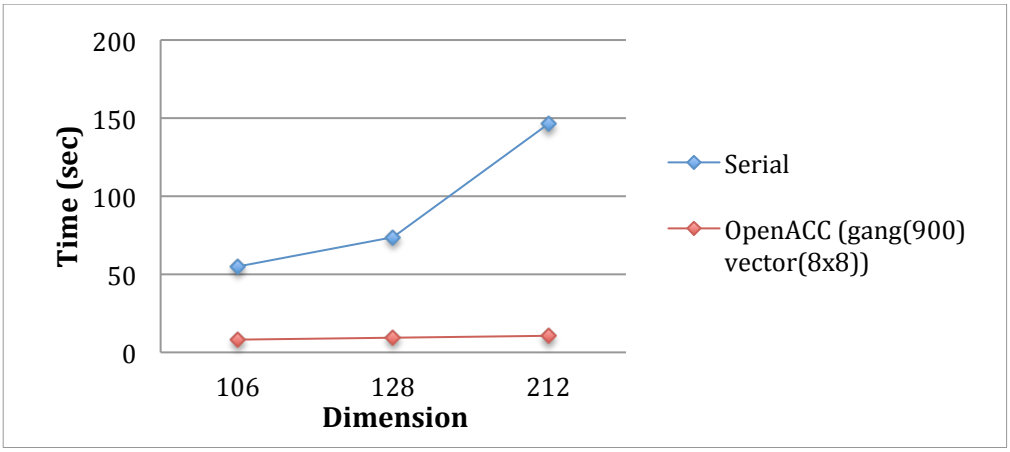

Figure 5 – Runtimes of the main loop in `spme_forces` for different lattice vector dimensions

5. Interoperability of OpenACC and CUDA

This section considers the `link_cell_pairs` subroutine of DL_POLY_4. It is one of the core components in DL_POLY and one of the most time consuming parts of the overall execution (See Figure 1 and 2.). Thus, it is a key point to accelerate the runtime of this component when the performance of the code is of interest. The `link_cell_pairs` subroutine constructs the Verlet neighbour list using link cells in DL_POLY. After the list is built, certain atom pairs are excluded from the list [5]. These two parts were accelerated in the CUDA port of DL_POLY as two consecutive steps. For the original CUDA implementation, there was a speedup of about 45x for this component [6]. After DL_POLY_4.01.1, there has been a major change in the source code of `link_cell_pairs_remove_exclusions`. However, the current CUDA port doesn't mirror this change.

The CUDA port of DL_POLY is a mix of very low-level C and C++ code exploiting architectural features of GPUs by using software-managed memory as well as using 3D grids and thread blocks. A different data structure was used in order to exhibit better efficiency. Moreover, all of the CUDA kernels were designed to minimise data transfer between CPU and GPU. It is a very sophisticated set of CUDA kernels and it requires a significant amount of time to maintain the code base. For this reason, the next section considers an OpenACC port of the `link_cell_pairs_remove_exclusions` component along with integrating it with the rest of the CUDA kernels. In this section, we test the effectiveness of running OpenACC interoperable with CUDA.

5.1. Early Performance Results

Given the excluded atoms list, the main loop illustrated in Table 8 constructs the latest version of the neighbour list by excluding the required interactions. There is an explicit data dependency in the inner kk-loop when updating the list and m_end variable. In the original CUDA port, this was handled by efficient usage of shared memory and `atomicSub()/atomicAdd()` functions of the CUDA programming model. On top of this, the kk^{th} iteration space was fully parallelised across the x^{th} thread block dimension. However, OpenACC doesn't allow the application developer to control the usage of memory and the PGI compiler used for this work doesn't yet support `atomic` clauses introduced in OpenACC 2.0 [10]. Thus, the "`!$acc loop seq`" directive is used to enforce the sequential ordering.

The outer loop is parallelised over all the atoms using "`!$acc kernels loop`" together with the `independent` clause. Once again, the number of gangs and vector lengths were inspired by the CUDA port. In terms of code refactoring, some reorganisation was applied to move the main computation in order to avoid branching. Furthermore, the `match()` function was inlined in the OpenACC version since PGI v13.9 doesn't yet support the `routine` clause introduced in OpenACC 2.0.

For this set of experiments, the data set TEST3 was used. First, the CUDA port of `link_cell_pairs` was enabled. Subsequently, the serial and OpenACC versions of the `remove_exclusions` were executed respectively. Performance tests indicated that there is a 1.1x decrease in the speedup of the overall execution time for the OpenACC implementation. This can be attributed to data transfers between the CPU and GPU memory.

```
Do i=1,natms
  l_end=list(0,i)
  m_end=l_end

  ii=lexatm(0,i)
  If (ii > 0) Then
    Do kk=l_end,1,-1
    j =list(kk,i)
    jj=ltg(j)
      If (match(jj,ii,lexatm(1:ii,i))) Then
        If (kk < m_end) Then
          list(kk,i)=list(m_end,i)
          list(m_end,i)=j
        End If
        m_end=m_end-1
      End If
    End Do
  End If
  list(-1,i)=list(0,i)
  list( 0,i)=m_end
End Do
```

```
!$acc kernels loop independent gang(900)
vector(64)
Do i=1,natms

  ii=lexatm(0,i)
  If (ii <= 0) cycle

  l_end=list(0,i)
  list(-1,i)=l_end
  m_end=l_end

  !$acc loop seq
  Do kk=l_end,1,-1
    …
    !Inline the function match()
    …
  End Do
  !$acc end loop

list(0,i)=m_end
End Do
!$acc end kernels loop
```

Table 8 – Sketch of `link_cell_pairs_remove_exclusions` component and its OpenACC port

In order to avoid the data transfer between the GPU and CPU, the `deviceptr` clause of OpenACC is used to access the data that has been already allocated and updated in the GPU memory that enables the integration of OpenACC with CUDA. In particular, the list is constructed on the GPU using the `link_cell_pairs` CUDA code. So, there is no need to transfer the list back and forth between host and device for removing the atoms in the exclusion list. For this purpose, a C version of the original FORTRAN code was written and integrated Fortran code using ISO-C bindings. Performance tests showed that there is an approximate of 2x speedup of the overall execution time due to minimising data transfers by use of `deviceptr`.

```
#pragma acc data deviceptr(list, lexatm, ltg)
{
#pragma acc kernels loop independent gang(900) vector(64)
for(i=1; i<=natms; i++){

  ii=lexatm[(i-1)*(mxexcl+1)];
  if(ii<=0) continue;

  l_end=list[(i-1)*(mxlist+3)+(2)];
  list[((i-1)*(mxlist+3))+1]=l_end;
  m_end=l_end;

  #pragma acc loop seq
  for(kk=l_end; kk>=1; kk--){
    j=list[((i-1)*(mxlist+3))+(kk+2)];
    jj=ltg[j-1];
    Match=0;
    if(lexatm[((i-1)*(mxexcl+1))+ii]>=jj){
      for(u=1; u<=ii; u++){
        LEXATM=lexatm[((i-1)*(mxexcl+1))+u];
        if(LEXATM>=jj){
          if(LEXATM==jj){
            Match=1;
            break;
          }
        }
      }
    }
    if(Match==1){
      if(kk<m_end){
        list[((i-1)*(mxlist+3))+(kk+2)]=list[((i-1)*(mxlist+3))
+(m_end+2)];
        list[((i-1)*(mxlist+3))+(m_end+2)]=m_end;
```

```
        }
      m_end=m_end-1;
      }
    list[((i-1)*(mxlist+3))+2]=m_end;
    }
  }
  }
```

Table 9 – The use of `deviceptr` for the OpenACC port of `link_cell_pairs_remove_exclusions` component

6. Conclusion

This work was an early evaluation of the OpenACC programming model for two main subroutines of the molecular dynamics software package DL_POLY. For `ewald_real_forces`, major code refactoring was required to enable OpenACC. A performance increase was obtained incrementally between different OpenACC implementations. However, the OpenACC port did not perform better than the serial execution time of the application. For `spme_forces`, adding a set of compiler directives with CUDA-inspired scheduling parameters demonstrated a significant performance increase. After handling data dependencies with the `reduction` clause, the rest of the code was suitable for many-core parallelisation and SIMD-style execution. However, the performance proved not to be as good as the CUDA version. The limitation here was that OpenACC doesn't allow the developer to utilise the GPU shared memory as it was exploited in the CUDA port.

This work was also concerned with the integration of OpenACC with the current CUDA port in order to maintain the GPU port of DL_POLY. Since OpenACC provides a high-level and simple programming model, it is easy to keep up with the changes of future DL_POLY versions. In this context, the role of the `deviceptr` clause was highlighted. However, the scalar nature of the algorithm was a limiting factor in the performance. New properties introduced in OpenACC 2.0 have great potential for better handling of the issues arising here.

Our view is that although OpenACC is still in its infancy, we feel that the features it strives to offer, namely programmability, portability and maintainability make it a promising tool for enabling large-scale complex applications on the road to Exascale. One feature that we think OpenACC would particularly benefit from is the ability for finer-grained memory management. Indications from GTC 2014 suggest that such features will be supported in OpenACC 3.0.

References

[1] http://www.stfc.ac.uk/CSE/randd/ccg/software/DL_POLY/25526.aspx.
[2] W. Smith and I. T. Todorov, A short description of DL_POLY, Molecular Simulation, Vol. 32, Iss.12-13, 2006.
[3] W. Smith, Molecular dynamics on hypercube parallel computers, Comput. Phys. Comm., Vol. 62, Iss. 2-3, pp. 229-248, 1991.
[4] I. T. Todorov and W. Smith, DL_POLY_3: The CCP5 National UK Code for Molecular Dynamics Simulation, Phil. Trans. Series A, 362:1822, pp. 1835-52, 2004.
[5] www.ccp5.ac.uk/DL_POLY/MANUALS/USRMAN4.pdf.
[6] C. Kartsakalis, R. Nestor, W. Smith and I. T. Todorov, Porting the DL POLY molecular dynamics package to GPGPUs, Workshop on GPUs and Accelerators in HPC, September 2010.
[7] M. Lysaght, M. Uchronski, A. Kwiecien, M. Gebarowski, P. Nash, I. Girotto and I. T. Todorov, Benchmarking and analysis of DL_POLY 4 on GPU clusters, PRACE whitepaper, PRACE-1IP, pdf: http://www.prace- ri.eu/IMG/pdf/benchmarking_and_analysis_of_dl_poly_4_on_gpu_clusters.pdf.
[8] A. Sunderland, S. Pickles, M. Nikolic, A. Jovic, J. Jakic, V. Slavnic, I. Girotto, P. Nash and M. Lysaght, An Analysis of FFT Performance in PRACE Application Codes, PRACE whitepaper (2012), pdf: http://www.prace-project.eu/IMG/pdf/An_Analysis_of_FFT_Performance_in_PRACE_Application_Codes.pdf.
[9] M. Uchronski, M. Gebarowski and A. Kwiecien, Optimization of SHAKE and RATTLE Algorithms, PRACE whitepaper, PRACE-2IP, pdf: http://www.prace-ri.eu/IMG/pdf/ optimization_of_shake_and_rattle_algorithms.pdf.
[10] http://www.openacc.org.
[11] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson and S. A. Jarvis, Accelerating Hydrocodes with OpenACC, OpeCL and CUDA, High Performance Computing, Networking, Storage and Analysis (SCC), pp.465-471, 2012.

[12] J. M. Levesque, R. Sankaran and R. Grout, Hybridizing S3D into an Exascale Application Using OpenACC: An Approach for Moving to Multi-petaflops and Beyond, High Performance Computing, Networking, Storage and Analysis (SCC), pp.15:1-15:11, 2012.
[13] S. Wienke, P. Springer, C. Terboven and D. an Mey, OpenACC - First Experiences with Real-World Applications, Proceedings of the 18th International Conference on Parallel Processing (Euro-Par'12), pp. 859-870, 2012.
[14] T. Hoshino, N. Maruyama, S. Matsuoka and R. Takaki, CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application, 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid), pp.136-143, 2013.
[15] B. Eagan and G. Civario, Investigating Performance Benefits from OpenACC Kernel Directives, PRACE whitepaper (2013), pdf: http://www.prace-project.eu/IMG/pdf/wp64_investigating_openacc.pdf.
[16] PRACE deliverable D7.2.1, A Report on the Survey of HPC Tools and Techniques, PRACE-3IP, pdf: www.prace-project.eu/IMG/pdf/d7.2.1.pdf.
[17] http://www.uio.no/english/services/it/research/hpc/abel/more.
[18] https://www.ichec.ie/infrastructure/fionn

Acknowledgements

This work was financially supported by the PRACE-3IP project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-312763. The work was carried out in collaboration with NTNU using the Abel cluster. We would like to thank UNINETT Sigma for providing us access to Abel and Bjørn Lindi for his assistance. We also would like to thank Dr. Ilian T. Todorov for fruitful discussions during the course of this project.